AgentTorch: a Framework for Agent-based Modeling with Automatic Differentiation

Anonymous Authors

Abstract

Agent-based models (ABMs) are discrete simulators comprising agents that can act and interact in a computational world. ABMs are relevant across several disciplines as these agents can be cells in bio-electric networks, humans in the physical world, or even AI avatars in a digital ecosystem. Despite wide applicability, research in ABMs has been extremely fragmented and has not benefited from modern computational advances, especially automatic differentiation. This paper presents **AgentTorch**: a framework to design, simulate, and optimize agent-based models. AgentTorch definition can be used to build stochastic, non-linear ABMs across digital, biological, and physical realms; while ensuring gradient flow through all simulation steps. AgentTorch simulations are fully tensorized, execute on GPUs and can range from a few hundred agents in synthetic grids to millions of agents in real-world contact graphs. The end-to-end differentiability of AgentTorch enables automatic differentiation of simulation parameters and integration with deep neural networks (DNNs) in several ways, for both supervised and reinforcement learning. We validate AgentTorch through multiple case studies that study cell morphogenesis over bio-electric networks, infection disease epidemiology over physical networks and opinion dynamics over social networks. **AgentTorch** is designed to be a viable toolkit for scientific exploration and real-world policy decision-making. We hope AgentTorch can help bridge research in AI and agent-based modeling.

1 Introduction

Agent-based models (ABMs) [8] are discrete simulators that comprise a collection of agents which can act and interact within a computational world. They can explicitly represent the heterogeneity of an interacting population via underlying contact networks and model the adaptability of individual agent behavior for more realistic simulations. This enables domains experts to ground simulations in mechanistic understanding and explore the emergent effects of agent behavior and external interventions. ABMs are used to simulate heterogeneous systems across biological [22, 29, 21], physical [16, 5, 37], digital [20, 2, 28] and financial [24, 30] realms. For instance, ABMs have helped simulate: i) cells in a tumor micro-environment to evaluate antibody treatments for tumor suppression [22], ii) diseased humans in the physical world to decide lockdown strategies [27] and prioritize vaccination schedules [35], iii) avatars in a digital environment to counter misinformation [9] and vaccine hesitancy [2] and iv) firms in a financial network to predict housing market crashes [30]. Despite wide applicability, the adoption of ABMs for practical decision making has been scarce which can largely be attributed to computational constraints.

The utility of ABMs for practical decision depends upon their ability to recreate realistic populations with great detail, integrate with real-world data streams and analyze sensitivity of results. The simulation outputs are sensitive to scale of the input population [11], initial conditions [8] and model specification [12]. However, conventional ABM frameworks [26, 41] are very slow to execute, difficult to scale to million-size populations, tough to calibrate and largely rely on simple hand-crafted rules. Prior works have sought to alleviate performance constraints though super-computing clusters [7] or highly-customized C++ code [23]. However, these are difficult to implement and

Preprint. Under review.

	Cell Morphogenesis (Biological)	Spatial Epidemiology (Physical)	Opinion Dynamics (Digital)
Agents	Cells	Citizens	Marketers, Consumers
Objects	N/A	Virus (R0, gen_time) Pubs (lat-long, capacity)	Products (quality, cost)
Environment	2D grid space (cell-cell)	Real-world contact graph (citizen-pub, citizen-citizen)	Random graph (consumer-consumer)
Scale	400 agents; 1000 steps	1.5 million agents; 180 steps	8100 agents; 100 steps
Substeps	EvolveCell	TwoDoseVaccination NewTransmission SEIRMProgression	PurchaseProduct
Learnable	*Substep Transition*	*Object properties*	*Substep Policy*
Technique	Embed DNN inside ABM (code in Figure 3)	End-to-End integrate DNN with ABM (code in Figure 4)	Optimize scalar ABM parameters using auto-diff (code in Figure 5)
Algorithm	Supervised Learning	Supervised Learning	Reinforcement Learning

Figure 1: AgentTorch can be used to define diverse ABMs across biological, digital and physical realms; execute million-scale simulations; and support learning capabilities through automatic differentiation of simulation parameters and by integration with DNNs.

generalize; and don't benefit from modern advances in data-driven analytics. Motivated from parallel efforts in differentiable scientific computation for molecular dynamics [36, 18], computational chemistry [39] and physics engines [17], some recent works has sought to achieving highly performant ABMs by leveraging capabilities of automatic differentiation. Differentiable ABMs [16, 4] have shown promising results to accelerate simulations [15], calibrate by integrating with DNNs [16], conduct sensitivity analyzes with gradients [33] and learn more expressive rule sets using DNN parameterizations [29]. However, these benefits have been restricted to few specific examples and no general framework exists. Alleviating this gap is the focus of this work.

We introduce AgentTorch: a framework to define, simulate and optimize ABMs with automatic differentiation. First, AgentTorch takes a functional view to ABMs and introduces a model definition which can be used to build non-linear stochastic ABMs across across biological, physical and digital realms; while ensuring gradient flow through all simulation steps. Second, AgentTorch leverages the capabilities of accelerators such as GPUs/TPUs and automatic differentiation to achieve highly performant simulations, while abstracting the user from the engineering complexity. AgentTorch simulations are fully tensorized and can range from a few hundred agents in a synthetic environment to millions of agents interacting over real-world contact graphs. Third, AgentTorch models are fully-differentiable which enables the use of automatic differentiation to optimize ABM parameters and integrate with DNNs to extend capabilities of rule-based ABMs. Here, we validate the AgentTorch design through diverse case studies that span cell morphogenesis over bio-electric networks, infection disease epidemiology over physical networks and opinion dynamics over social networks.

2 AgentTorch: define, simulate and optimize agent-based models

AgentTorch is used to describe discrete-event simulations with multiple interacting agents. First, we introduce the design specification for defining an AgentTorch model. Second, we describe the implementation primitives and the system architecture for executing an AgentTorch simulation. Third, we describe learning with AgentTorch which leverages the benefits of automatic differentiation.

2.1 Model Definition

Definition 1 (AgentTorch model)**.** *An AgentTorch model is defined by the following tuple:* $\langle \mathcal{S}, \mathcal{G} \rangle$*. Here* $\mathcal{S} = \langle \mathcal{S}_{Ag}, \mathcal{S}_{Ob}, \mathcal{S}_{Env} \rangle$*, represents the set of states of the agents, objects and the environment respectively, including the agent-agent and agent-object interaction networks which form a part of*

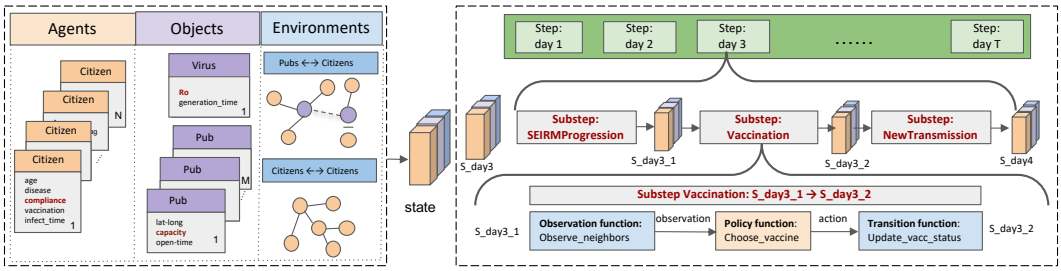

Figure 2: **Defining an AgentTorch model for spatial epidemiology**. The simulation has N citizen that interact through direct mobility and co-locate across pubs (Object) to spread the virus (Object). The simulator state is a collection of properties that describe each of these entities, is initialized once and transformed during T steps of simulation. Each step models the disease progression of infected agents, vaccination of susceptible agents and transmission of new infections, to recursively transform the simulation state over multiple substeps. AgentTorch is designed to ensure gradient flow through all simulation steps and enables automatic differentiation of any state property or substep function.

the state of the environment. $\mathcal{G}$ represents the set of substeps, where each substep is composed of the three functions – i) the observation function which generates objects for all agents, $o : \mathcal{S} \rightarrow \mathcal{O}$, where $\mathcal{O}$ is the space of all observations, ii) the policy function which generates the actions taken by agents, $\pi : \mathcal{H} \rightarrow \mathcal{A}$, where $\mathcal{H}$ is the set of trajectories of all historical observations and $\mathcal{A}$ is the set of all actions (over all agents) and iii) the transition function which captures the system evolution by generating the next state given the current state and the current action $t : \mathcal{S} \times \mathcal{A} \rightarrow \mathcal{S}$.

AgentTorch model has two components: i) state (S) ii) substep (G: $S \mapsto S$). The state is a collection of properties that describe different entities in the simulation. A substep is a unit operation that transforms the state to produce a new state. When reading the terminology below, consider the example in Figure 2 which simulates the spread of an infection disease (like covid-19).

AgentTorch model has three kinds of entities:

- *Agents* which observe, act and interact within a computational world. For instance, these can be infected citizens that spread diseases (with properties like {age, disease_stage}).
- *Objects* which interface between agents but don't have the agency to act. For instance, these can be a virus that carries infection (with properties like {RO}) or a pub where citizens co-locate (with properties like {lat-long, capacity}).
- *Environments* which facilitate the interactions of agents with other agents or objects. These are interaction graphs of one of two types: *agent-agent* and *agent-object*. For instance, a *citizen-pub* (agent-object) graph can represent interaction of citizens across different pubs in a geo-locality.

Each state property is initialized once to define the initial state and may be transformed during the simulation. The simulation comprises a series of *substeps* that recursively transform the state, in sequence. Each substep is the basic unit which transforms the state to produce a new state. To produce the next state for a given agent, a substep invokes three kinds of functions in sequence:

1. *Substep Observation* which uses the current state and returns an observation for the agents. For instance, an agent can receive an observation regarding the state of infection and vaccination status of its immediate neighbors (`observation = observe_neighbors(state)`).
2. *Substep Policy* which uses this generated observation, along with the entire history of earlier observations (compiled as `observation_history`) to yield the agents' actions. For instance, an agent uses its current observations of vaccination by neighbors and historical deaths to decide whether to vaccinate itself (`action = choose_vaccine(observation_history)`).
3. *Substep Transition* which uses the current state and agent actions to generate the next state. For instance, a non-vaccinated agent in contact with infected agents may get infected itself (`next_state = update_vacc_status(state, action)`).

2.2 Simulator Implementation

AgentTorch leverages the PyTorch API to describe simulation state (`nn.ParameterDict`) and the computational graph of substeps (`nn.ModuleDict`). Furthermore, this provides in-built support for tensorized GPU execution, automatic differentiation and streamlines integration of DNNs with mechanistic ABMs. Here, we describe the framework primitives and high-level modules.

Primitives AgentTorch has two primitives: variable `at_var` and method `at_function` which maps from `at_var` to `at_var`. `at_var` has a value, shape, dtype, learnable flag and extends `torch.nn.Parameter` class. Hence, this `at_var` can be a scalar, tensor or even a neural network. `at_function` has inputs, outputs and arguments each of which is an `at_var` and extends `torch.nn.Module` class. These form the building blocks for an AgentTorch model.

Each state property (`at_property`) has a value (instance of `at_var`) and an initialization function (instance of `at_function`). Each property is initialized once to compile the initial state (`at_state`). This can be transformed during the simulation or even learned via optimization by setting the flag `learnable=True`. The `at_state` is a collection of `at_property` instances which describe the agents, objects and environments; and extends `torch.nn.ParameterDict` class.

Each substep (`at_substep`) is a mapping from `at_state` to `at_state`. Each `at_substep` is a collection of `at_function` instances that describe the observation, policy and transition functions. These functions are invoked in sequence to produce the `next_state`. Specifically, i) `observation = substep.observation_function(state)`, ii) `action = substep.policy_function(observation)`, iii) `next_state = substep.transition_function(state, action)`. Any substep function can be independently learned during optimization by setting its arguments flag `learnable=True` and may also be parameterized with a deep neural network. This modularity allows embed neural networks inside mechanistic simulators and learn specific components without affecting the rest. Each substep extends `torch.nn.ModuleDict` class. The design helps AgentTorch simulations ensure gradient flow (and parameter tracking) through each substep and across all the steps of a simulation.

High-level modules AgentTorch has multiple high-level modules to define models, execute simulations and track variables. `Config` and `Runner` are exposed to the user for defining the model and executing episodes of the simulation. Internally, these interface with `Controller` to initialize the simulator state, sequence substeps, execute all episodes and track variables.

Model is defined by instantiating a `config = Config()`. This config enables creating agents and objects, inserting interaction environments, defining execution metadata and creating simulation substeps. The code listing below uses the `config` to create infectious citizens agents (line 5) and infecting virus object (line 6), define citizen mobility networks (line 8), describe a infection transmission substep (line 10) and execute the simulation for 10 episodes (line 12).

```
1  from AgentTorch import Config
2
3  config = Config()
4
5  config.add_agents(name="citizens", num_citizens, prop_list)
6  config.add_objects(name="virus", num_strains, prop_list)
7
8  config.add_environment(type="agent-agent", src="from_file", path="
       citizen_citizen.networkx")
9
10 conf.add_substeps(name="new_transmission", active_agents="citizens",
       transition=TransitionFunc)
11
12 config.add_metadata("num_episodes", 10)
```

Listing 1: Using the AgentTorch Config API to define a model

Simulation is executed by instantiating a `runner = Runner(config)`. Each `runner` has four modules: `runner.init`, `runner.step`, `runner.reset` and `runner.parameters`. First, `runner.init()` is used to initialize all state properties (`state`) and creates a tracking registry of substep functions (`registry_dict`) by invoking `controller.initialize()`.

Second, `runner.step()` is used to run all steps in a simulation episode. Each episode step invokes `controller.execute_substep(state, registry_dict)` to execute substeps in sequence. `controller.execute_substep` handles control flow dependency, tracks parameters and transforms the `state` by invoking: `observation = controller.observe(state)`, `action = controller.act(observation)` and `next_state = controller.progress(state, action)` in order. Third, `runner.reset()` is used to reinitialize the state of the simulator before the start of subsequent episodes. While the default is to just use `runner.init()`, this function is often overloaded to specify custom reset functions (as in Case Study 1 using state from prior episodes). Fourth, `runner.parameters()` tracks and return all learnable parameters in the simulation episode. These parameters can be properties of the simulator state or arguments of substep functions. This is then used to define optimizers, such as `torch.optim.SGD(runner.parameters())`. Finally, `runner.trajectory` tracks the simulation state across multiple steps and episodes and is used to define loss functions and plot outputs.

```python
import torch
import AgentTorch as AT

# Step 1: define entities, metadata and substeps
config = AT.Config()
config.add_agents(...)
config.add_objects(...)
config.add_environments(...)
config.add_metadata(...)
config.add_substeps(...)

# Step 2: create simulation instance
runner = AT.Runner(config)

# Step 3: initialize simulation state and create registry
runner.init()

# Step 4: create optimizer using learnable simulation parameters
opt = torch.optim.SGD(list(runner.parameters()), lr=config_lr)

for episode in range(num_episodes):
    opt.zero_grad()

    # Step 5: reset state before each episode
    runner.reset()

    # Step 6: execute all substeps in sequence
    runner.step(num_steps)

    # Step 7: read the trajectory to extract output
    trajectory = runner.trajectory
    output = generate_output(trajectory)

    # Step 8: compute loss and optimize parameters
    loss = loss_fn(output, ground_truth)
    loss.backward()
    opt.step()
```

Listing 2: Using the AgentTorch API to define simulate and optimize agent-based models

2.3 Gradient-based Optimization

AgentTorch is designed to ensure gradient-flow through all substeps of the simulation and is compatible with automatic differentiation. This allows using gradient-based learning to update properties of the state or arguments of any substep function. All learnable parameters across the simulator can be accessed via `runner.parameters()` and used in `torch.optim` to define custom optimizers.

AgentTorch supports both supervised learning (SL) and reinforcement learning (RL) using first-order gradient estimates [38] and leverages the PyTorch API for optimization. Broadly, there are three modes of optimization configurations:

```
config = Config()
config.add_substep(EvolveCell, transition=MyCustomDNN())

runner = Runner(config)

opt = Optim.SGD(list(runner.parameters()))

for i in range(num_episodes):
  opt.zero_grad()
  runner.reset()

  runner.step(num_steps)

  emergent_pattern = get_cell_states(runner.trajectory['last_state'])
  supervised_loss = nn.MSELoss()(emergent_pattern, true_pattern)

  supervised_loss.backward()
  opt.step()
```

Figure 3: **C1: Embed DNN inside ABM with AgentTorch**. [29] uses cellular automata to simulate morphogenesis and parameterizes the update rules with a CNN. AgentTorch is used to learn the cellular automata rules by representing the transition function of a substep with a DNN. Enabling this requires only a few additional lines as shown in the pseudocode on the left, and is an instance of *Mode 2* in sec 2.3. The experiment follows from [29] and more details are included in the appendix. Corresponding results in appendix A show the emergent pattern for two shapes (lizard and butterfly) at different steps along the simulation.

.

- *Mode 1: Optimize scalar/tensor ABM parameters*. For instance, this may involve calibrating the R0 parameter of a virus to death statistics using SL (sec 3.1) or learning purchase policy of an agents to maximize expected utility using RL (sec 3.3). The optimizer is defined as `torch.optim.SGD(runner.parameters())`.

- *Mode 2: Embed DNN inside ABM to learn substep functions*. For instance, an unknown mechanism can be parameterized with a neural network without affecting any other components of the simulator; and learned to reproduce observed simulation output using SL (sec 3.1). This neural substep function is defined in the `config` and its parameters tracked in `runner.parameters()`. The optimizer is defined as `torch.optim.SGD(runner.parameters())`..

- *Mode 3: Integrate ABM with DNN pipelines*. Instead of optimizing components of the simulation, AgentTorch model can define a proxy objective and provide gradients to learn an external black-box models (`external_nn`). For instance, they may be used to jointly forecast infections across multiple counties with distinct simulators via SL (sec 3.2). The hybrid optimizers can be defined with `torch.optim.SGD(list(external_nn.parameters() + runner.parameters()))`.

3 Case Studies

Here, we present diverse case studies to show the flexibility of AgentTorch in definition, simulation, and optimization. These case studies span digital, physical, and biological realms; scale from a few hundred agents in a synthetic grid spaces to millions of agents over city-scale contact networks; and involve learning simulation parameters, agent policies, and transition rules. Specifically, these include cells in a bio-electric micro-environment assembling organs, human citizens in a physical environment spreading infections and avatars in a digital environment sharing opinions. The flexibility in design is coupled with computational benefits realized by tensorization, GPU execution, and support for automatic differentiation which unlocks new capabilities via seamless (end-to-end) integration with deep neural networks. For this analysis, we benchmark previously introduced simulators using AgentTorch. The key objective is to demonstrate the capabilities of the design of AgentTorch which allows to specify diverse multi-agent scenarios, execute million-scale simulations, conduct gradient-based optimization and evaluate interventions in agent-based models.

3.1 Cell Morphogenesis

Morphogenesis is the process of an organism's shape development where cells interact over bio-electric networks to self-assemble into tissues and organs. The process is extremely robust to perturbations where several species have the ability to regenerate entire organs by repairing damage (to intermediate states) or produce viable organs even from atypical initial states. Understanding the mechanism behind behind morphogensis is an active area of research and key to progress in regenerative medicine. We follow from [29] which extends cellular automata (CAs) to identify cell-level rules that result in adaptive and robust morphogensis. CAs consist of a grid of cells that are iteratively updated with the same set of rules applied to each cell at every step. The new state of a cell depends only upon the state of a few cells in its immediate neighborhood. The goal of the simulation is two fold: a) learn the cell-level mechanism by representing it with a deep neural network, b) validate robustness of the learned mechanism to perturbations in initial state.

Following from [29], in the simulation, agents are cells with 16-dim property state. Agent-agent interaction are described over a 2D grid space environment. The state is initialized with a single active agent and produces a multi-cellular pattern through the simulation. The simulation has a single substep (`EvolveCell`) with transition function is parameterized with a CNN and describes how cells interact with neighbors to update their state. The output of the simulation is a 2D grid pattern of all cell states (denoting organism shape) and the learning objective is supervised loss against an expected shape (or grid pattern). The goal is to learn a transition function robust to perturbations in initial state and involves jointly optimizing over multiple simulations with varying initial states.

AgentTorch demonstrates two key capabilities:

- **C1**: AgentTorch allows to embed DNN inside an ABM. Here, the transition function of `EvolveCell` substep is parameterized with a CNN. This is captured by `runner.parameters()` and can be used with an optimizer as shown by psuedocode in Figure 3.
- **C2**: AgentTorch enables joint optimization and parameter sharing between multiple runners. Here, multiple `runner` objects created for different initial states utilize a shared optimizer and each is simulated via `runner[j].step_from_params(runner[i].substep.parameters())`. Pseudocode and results for this experiment are included in the appendix.

3.2 Spatial Infectious Disease Epidemiology

Infectious diseases spread through physical contact with infected agents and have a two-time scale nature of transmission of new infections and stage progression of already infected agents. Understanding the spread of infection is key to designing effective intervention policies. In the context of COVID-19, this involved deciding lockdown policies, prioritizing vaccination schedules and selecting testing strategies. In practice, such decisions are highly complex as they require considering scale of the population, stochastic behavior of individual agents as well as properties of the intervention. For instance, effective public health policies during COVID-19 included delaying administration of the second vaccine dose, prioritizing test speed over specificity. Evaluating these decisions in-silico requires granular and data-driven simulations, fast calibration and sensitivity analyzes. The goals of this simulation are: a) recreate real-world million-scale populations, b) improve calibration of simulation parameters using DNNs, c) analyze sensitivity of diverse interventions.

Following from [16, 35, 23, 15], in the simulation, agents are citizens with 5-dim property state (age, occupation, disease-stage, infected-time, vaccine-status) that spread covid-19 infection. Objects include both the infecting virus and co-location centers like pubs, schools and care-homes. Environments are obtained using real-world contact graphs and describes interactions in citizen-citizen mobility networks and citizen-pub co-location networks. The simulator state is initialized with a few infected agents. Each simulation step has two substeps `InfectionTransmission` and `SEIRMProgression` which describe transmission of new infections and an SEIRM progression of previously infected agents, respectively. Discrete stochasticity in the simulation is handled by reparameterizing with gumbel-softmax gradient estimator, to ensure differentiability. The output of the simulation is the histogram of citizen disease stages and the learning objective for calibration is a supervised loss against ground truth case statistics (from CDC).

AgentTorch demonstrates the following capabilities:

	Mechanism to calibrate simulation parameters	COVID-19			Influenza		
		ND	RMSE	MAE	ND	RMSE	MAE
a	Optimize directly in runner	2.39 ± 0.39	205.14 ± 42.56	73.66 ± 10.88	0.88 ± 0.14	2.97 ± 0.44	2.64 ± 0.43
b	Optimize in external_nn using gradient from the runner	1.15 ± 0.24	67.09 ± 23.89	35.50 ± 7.36	0.50 ± 0.19	1.78 ± 0.62	1.50 ± 0.57
c	Optimize in external_nn using joint gradient from multiple runners	0.97 ± 0.18	50.99 ± 12.12	30.02 ± 5.60	0.41 ± 0.02	1.47 ± 0.06	1.22 ± 0.06

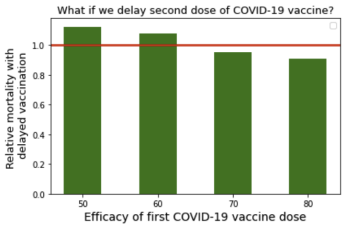

Figure 4: **C3-C5** AgentTorch can simulate ABMs with millions of agents, build hybrid DNN-ABM pipelines and jointly optimize multiple ABMs by changing only a few lines of code. AgentTorch is used to calibrate simulations with millions of agents and forecast spread of two diseases - COVID-19 and Influenza over three different learning situations denoted by (a), (b) and (c). This calibrated model is used to answer policy questions as shown on the right. These experiments follow from [16] and more details, with pseudocode for each learning situation are in the appendix

- **C3**: AgentTorch enables realistic simulations with million-scale populations and real-world contact networks, all while abstracted from the engineering complexity. The same API scales to millions of agents and can support real-world contact graphs generated offline. The run-time performance benchmarking is shown in the appendix.
- **C4**: AgentTorch enables integrating ABMs with DNNs. Here, this pipeline is used to calibrate simulation parameters using gradient-based learning by designing hybrid optimizers. Pseudocodes, corresponding to results in figure 4(left), are shown in figure 10 of appendix B.
- **C5**: AgentTorch allows flexible experimentation through its modular design. First, AgentTorch can be used to evaluate policy interventions through white-box scenario analysis. Figure 4 (right) shows results which evaluate the efficacy of delayed vaccination schedule. Second, AgentTorch can generalize across simulation assumptions by changing a few lines of code. Figure 4 (left) shows a model built for COVID-19 can be adapted to Influenza by just replacing a single substep (`SEIRMProgression` with `SIRSProgression`). More details are in the appendix.

3.3 Social Opinion Dynamics

Digital interactions are already ubiquitous and are become increasingly becoming more relevant with the advent of autonomous agents. Such agents, trained to act strategically, will become integral to society and business as they redefine interfaces with humans to mitigate vaccine hesitancy against diseases, advertise new products in competitive markets etc. Typically agents in these systems interact in two ways – directly via communicating with each other (influenced by their individual follower tendencies) and indirectly via affecting the environment or objects. We implement a standard opinion dynamics model, focusing on direct interactions, to simulate lock-in of consumer behavior. The goal of this simulation is: a) learn agent policies that maximize utility over time horizon.

Following from [20], in the simulation, consumers and marketers are the two types of agents. Objects include products that the marketers advertise to consumers. We specifically consider a duopoly with two products. Environment describes interactions through consumer-consumer networks that are defined using grid graphs in the simulation. The simulation has a single substep `PurchaseProduct` where agents observe purchase behavior of neighbors to make a discrete purchase decision. The output of the simulation is the observed utility for all agents and the learning objective is to maximize each agent's expected utility over the finite time horizon. The discrete stochasticity in the simulation, arising from agent purchase behavior, is reparameterized with the gumbel softmax gradient estimator allowing for automatic differentiation through time. In principle, a score-function gradient estimate (some variant of REINFORCE) can also be used. We note that the goal is to demonstrate ability to use AgentTorch for sequential decision making irrespective of the specific choice of algorithm.

AgentTorch demonstrates the following capabilities:

- **C6**: AgentTorch allows learning agent policies by specifying custom reward functions. End-to-end differentiability of AgentTorch enables reinforcement learning with first-order policy gradients [40]. Pseudocode and specific example in context of opinion dynamics is shown in Figure 5.

```
config = Config()
runner = Runner(config)

opt = Optim.SGD(list(runner.parameters()))

for i in range(num_episodes):
  opt.zero_grad()
  runner.reset()

  runner.step(num_steps)

  episode_return = get_agents_utility(runner.trajectory)
  policy_loss = -episode_return

  policy_loss.backward()
  opt.step()
```

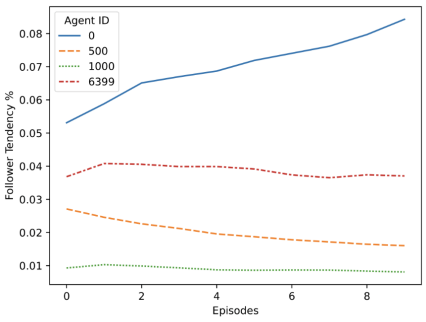

Figure 5: **C6 - Policy optimization with AgentTorch**. In social opinion dynamics [20], purchase behavior is parameterized with a follower tendency and agents learn personalized optimal follower tendencies to maximize utilities over timesteps. This policy learning is enabled easily in AgentTorch with parameters in `runner` as shown in the pseudocode on the left. The learning curve with follower tendencies versus episodes for a few sample agents is shown on the right. The experimental setup follows from [20] and more details are in the appendix.

4 Related Work

Automatic differentiation is becoming an integral part of scientific computation with recent applications in computational finance [13], atmospheric modeling [14], protein modeling [1, 25], computational chemistry [39], physics engines [17, 19]. Hence, in recent times, several frameworks have emerged with the goal to equip automatic differentiation libraries with simulation capabilities. Some examples include JAX.MD [36] and TorchMD [18] for molecular dynamics, TorchDyn [32] for neural differential equations, JAXFluids [6] for fluid dynamics. These packages have helped realize highly performance simulations by leveraging the capabilities of XLA and integrations with deep learning models. Some recent work has explored differentiable agent-based modeling [16, 4] with promising results to accelerate simulations [15], improve calibration by integrating with deep neural networks [16, 34], conduct one-shot sensitivity analysis using gradients [33], and replace mechanistic rules with neural networks [29, 31]. However, this has been restricted to very specific examples and no general framework exists. Designing such a framework for agent-based modeling presents unique challenges due to the presence of stochasticity of individual behavior, heterogeneity of population interactions, multi-scale evolution, nested conditional rule sets and extremely diverse applicability. [3] integrate ABMs with DNNs using 0th-order gradients as the simulators are not tensorized or differentiable. We aim to fill the gap with AgentTorch. AgentTorch can be used to build extremely diverse simulators across digital, physical, and biological realms and supports learning with automatic differentiation, as demonstrated by our analysis.

5 Conclusions and Future Work

We introduce AgentTorch: a framework to define, simulate and optimize agent-based models (ABMs). First, the AgentTorch model definition has been used to design diverse ABMs across biological, digital and physical realms. Second, The simulations are fully tensorized, execute on GPUs and can range from few hundred agents in synthetic grids to millions of agents over real-world contact graphs. Third, AgentTorch models are designed to be fully differentiable which enables use of automatic differentiation of all simulation parameters and integrate with DNNs in several ways to extend capabilities of mechanistic ABMs. For future work, we plan to build an AgentTorch gym, similar to [10], to provide a test bed of real-world simulators for AI researchers. AgentTorch is designed to be a viable toolkit for scientific exploration as well as real-world policy decision-making. We hope AgentTorch can help bring research in AI and agent-based models closer together.

References

[1] Mohammed AlQuraishi. End-to-end differentiable learning of protein structure. *Cell systems*, 8(4):292–301, 2019.

[2] Camilla Ancona, Francesco Lo Iudice, Franco Garofalo, and Pietro De Lellis. A model-based opinion dynamics approach to tackle vaccine hesitancy. *Scientific Reports*, 12(1):11835, 2022.

[3] Leo Ardon, Jared Vann, Deepeka Garg, Thomas Spooner, and Sumitra Ganesh. Phantom-a rl-driven multi-agent framework to model complex systems. In *Proceedings of the 2023 International Conference on Autonomous Agents and Multiagent Systems*, pages 2742–2744, 2023.

[4] Gaurav Arya, Moritz Schauer, Frank Schäfer, and Christopher Rackauckas. Automatic differentiation of programs with discrete randomness. *Advances in Neural Information Processing Systems*, 35:10435–10447, 2022.

[5] Joseph Aylett-Bullock, Carolina Cuesta-Lazaro, Arnau Quera-Bofarull, Miguel Icaza-Lizaola, Aidan Sedgewick, Henry Truong, Aoife Curran, Edward Elliott, Tristan Caulfield, Kevin Fong, et al. June: open-source individual-based epidemiology simulation. *Royal Society open science*, 8(7):210506, 2021.

[6] Deniz A Bezgin, Aaron B Buhendwa, and Nikolaus A Adams. Jax-fluids: A fully-differentiable high-order computational fluid dynamics solver for compressible two-phase flows. *Computer Physics Communications*, 282:108527, 2023.

[7] Keith R Bisset, Jiangzhuo Chen, Xizhou Feng, VS Anil Kumar, and Madhav V Marathe. Epifast: a fast algorithm for large scale realistic epidemic simulations on distributed memory systems. In *Proceedings of the 23rd international conference on Supercomputing*, pages 430–439, 2009.

[8] Eric Bonabeau. Agent-based modeling: Methods and techniques for simulating human systems. *Proceedings of the national academy of sciences*, 99(suppl_3):7280–7287, 2002.

[9] Julii Brainard, PR Hunter, and Ian R Hall. An agent-based model about the effects of fake news on a norovirus outbreak. *Revue d'epidemiologie et de sante publique*, 68(2):99–107, 2020.

[10] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. OpenAI Gym. *arXiv preprint arXiv:1606.01540*, 2016.

[11] Antonio Bru, E Alós, JC Nuño, and M Fernández de Dios. Scaling in complex systems: a link between the dynamics of networks and growing interfaces. *Scientific reports*, 4(1):1–7, 2014.

[12] Patrick Cannon, Daniel Ward, and Sebastian M Schmon. Investigating the impact of model misspecification in neural simulation-based inference. *arXiv preprint arXiv:2209.01845*, 2022.

[13] Luca Capriotti. Fast greeks by algorithmic differentiation. *Available at SSRN 1619626*, 2010.

[14] Gregory R Carmichael, Adrian Sandu, et al. Sensitivity analysis for atmospheric chemistry models via automatic differentiation. *Atmospheric Environment*, 31(3):475–489, 1997.

[15] Ayush Chopra, Esma Gel, Jayakumar Subramanian, Balaji Krishnamurthy, Santiago Romero-Brufau, Kalyan S Pasupathy, Thomas C Kingsley, and Ramesh Raskar. Deepabm: scalable, efficient and differentiable agent-based simulations via graph neural networks. In *Winter Simulation Conference (WSC)*, 2021.

[16] Ayush Chopra, Alexander Rodriguez, Jayakumar Subramanian, Balaji Krishnamurthy, B Aditya Prakash, and Ramesh Raskar. Differentiable agent-based epidemiology, 2023.

[17] Filipe de Avila Belbute-Peres, Kevin Smith, Kelsey Allen, Josh Tenenbaum, and J Zico Kolter. End-to-end differentiable physics for learning and control. *Advances in neural information processing systems*, 31, 2018.

[18] Stefan Doerr, Maciej Majewski, Adrià Pérez, Andreas Kramer, Cecilia Clementi, Frank Noe, Toni Giorgino, and Gianni De Fabritiis. Torchmd: A deep learning framework for molecular simulations. *Journal of chemical theory and computation*, 17(4):2355–2363, 2021.

[19] C. Daniel Freeman, Erik Frey, Anton Raichuk, Sertan Girgin, Igor Mordatch, and Olivier Bachem. Brax - a differentiable physics engine for large scale rigid body simulation, 2021.

[20] Michael Garlick and Maria Chli. Agent-based simulation of lock-in dynamics in a duopoly. In *9th International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 1545–1546, 2010.

[21] Voit EO Glen CM, Kemp ML. Agent-based modeling of morphogenetic systems: Advantages and challenges. *PLoS Computational Biology*, 2019.

[22] Chang Gong, Oleg Milberg, Bing Wang, Paolo Vicini, Rajesh Narwal, Lorin Roskos, and Aleksander S Popel. A computational multiscale agent-based model for simulating spatio-temporal tumour immune response to pd1 and pdl1 inhibition. *Journal of the Royal Society Interface*, 14(134):20170320, 2017.

[23] Robert Hinch, William JM Probert, Anel Nurtay, Michelle Kendall, Chris Wymant, Matthew Hall, Katrina Lythgoe, Ana Bulas Cruz, Lele Zhao, Andrea Stewart, et al. Openabm-covid19—an agent-based model for non-pharmaceutical interventions against covid-19 including contact tracing. *PLoS computational biology*, 17(7):e1009146, 2021.

[24] Cars H Hommes. Modeling the stylized facts in finance through simple nonlinear adaptive systems. *Proceedings of the National Academy of Sciences*, 99(suppl_3):7221–7228, 2002.

[25] John Ingraham, Adam Riesselman, Chris Sander, and Debora Marks. Learning protein structure with a differentiable simulator. In *International Conference on Learning Representations*, 2019.

[26] Jackie Kazil, David Masad, and Andrew Crooks. Utilizing python for agent-based modeling: The mesa framework. In Robert Thomson, Halil Bisgin, Christopher Dancy, Ayaz Hyder, and Muhammad Hussain, editors, *Social, Cultural, and Behavioral Modeling*, pages 308–317, Cham, 2020. Springer International Publishing.

[27] Cliff C Kerr, Robyn M Stuart, Dina Mistry, Romesh G Abeysuriya, Katherine Rosenfeld, Gregory R Hart, Rafael C Núñez, Jamie A Cohen, Prashanth Selvaraj, Brittany Hagedorn, et al. Covasim: an agent-based model of covid-19 dynamics and interventions. *PLOS Computational Biology*, 17(7):e1009149, 2021.

[28] Ian S Lustick et al. Agent-based modelling of collective identity: testing constructivist theory. *Journal of Artificial Societies and Social Simulation*, 3(1):1, 2000.

[29] Alexander Mordvintsev, Ettore Randazzo, Eyvind Niklasson, and Michael Levin. Growing neural cellular automata. *Distill*, 2020.

[30] Federico Guglielmo Morelli, Michael Benzaquen, Marco Tarzia, and Jean-Philippe Bouchaud. Confidence collapse in a multihousehold, self-reflexive dsge model. *Proceedings of the National Academy of Sciences*, 117(17):9244–9249, 2020.

[31] Elias Najarro, Shyam Sudhakaran, Claire Glanois, and Sebastian Risi. Hypernca: Growing developmental networks with neural cellular automata. *ICLR Workshop on Cells to Societies*, 2022.

[32] Michael Poli, Stefano Massaroli, Atsushi Yamashita, Hajime Asama, and Jinkyoo Park. Torchdyn: A neural differential equations library. *arXiv preprint arXiv:2009.09346*, 2020.

[33] Arnau Quera-Bofarull, Ayush Chopra, Joseph Aylett-Bullock, Carolina Cuesta-Lazaro, Anisoara Calinescu, Ramesh Raskar, and Michael Wooldridge. Don't simulate twice: One-shot sensitivity analyses via automatic differentiation. In *Proceedings of the 2023 International Conference on Autonomous Agents and Multiagent Systems*, pages 1867–1876, 2023.

[34] Arnau Quera-Bofarull, Ayush Chopra, Anisoara Calinescu, Michael Wooldridge, and Joel Dyer. Bayesian calibration of differentiable agent-based models. *ICLR Workshop on AI for Agent-based Models*, 2023.

[35] Santiago Romero-Brufau, Ayush Chopra, Alex J Ryu, Esma Gel, Ramesh Raskar, Walter Kremers, Karen S Anderson, Jayakumar Subramanian, Balaji Krishnamurthy, Abhishek Singh, et al. Public health impact of delaying second dose of bnt162b2 or mrna-1273 covid-19 vaccine: simulation agent based modeling study. *bmj*, 373, 2021.

[36] Samuel Schoenholz and Ekin Dogus Cubuk. Jax md: a framework for differentiable physics. *Advances in Neural Information Processing Systems*, 33:11428–11441, 2020.

[37] Neal R Smith, James M Trauer, Manoj Gambhir, Jack S Richards, Richard J Maude, Jonathan M Keith, and Jennifer A Flegg. Agent-based models of malaria transmission: a systematic review. *Malaria journal*, 17(1):1–16, 2018.

[38] Hyung Ju Suh, Max Simchowitz, Kaiqing Zhang, and Russ Tedrake. Do differentiable simulators give better policy gradients? In *International Conference on Machine Learning*, pages 20668–20696. PMLR, 2022.

[39] Teresa Tamayo-Mendoza, Christoph Kreisbeck, Roland Lindh, and Alán Aspuru-Guzik. Automatic differentiation in quantum chemistry with applications to fully variational hartree–fock. *ACS central science*, 4(5):559–566, 2018.

[40] Nina Wiedemann, Valentin Wüest, Antonio Loquercio, Matthias Müller, Dario Floreano, and Davide Scaramuzza. Training efficient controllers via analytic policy gradient. *arXiv preprint arXiv:2209.13052*, 2022.

[41] Uri Wilensky. Netlogo. http://ccl.northwestern.edu/netlogo/, Center for Connected Learning and Computer-Based Modeling, Northwestern University, Evanston, IL, 1999.